DESIGN DOCUMENT

Author: Niland Sshumacher
Student ID: 1612833
email:naschuma@ucsc.edu

PROJECT DESCRIPTION

This proxy http server to run on linux implements a file cache which can store a specified number of files, and a replacement policy as program flags. The proxy server uses HTTP to interact with both client and server and is designed to be left on. The proxy server was tested using curl, and could possibly work with other programs using http/1.1.

PROGRAM LOGIC

The program must be passed two mandatory port fields, and two optional cache size, and replacement policy flags. The server first creates a client socket using the server port and then listens for connecting clients on the client port. Whenever a client request is sent, the proxy forwards that request to the server, and forwards any responses from the server back to the client.

During a GET request, the proxy will store the file returned by the server into a cache in memory. If the requested file is in the cache the proxy will first send a HEAD request to the server to check if the file is obsolete using the 'Last Modified' header. If the file is not obsolete, and is saved in the file cache then the proxy will send the client the cached file, instead of sending a get request to the server. This caching method can save much time, by limiting the amount of data needed to be sent over the network.

On default 3 files can be stored in the cache, but this can be changed using the -c flag followed by the cache size as an argument. The replacement policy used by the cache on default is FIFO, but can be changed to a least recently used replacement policy with the -u flag.

Unlike the previous project, this server is only singly threaded, and can only handle one client at a time.

DATA STRUCTURES

Httpserver uses two 4kb buffers to send and receive http headers and to read and write files. The program stores cached files in a singly linked list of buffers on the heap.

FUNCTIONS

- void handle_connection(int connfd)
  Enters an infinite loop which waits to receive data from the accepted socket. Upon receiving data it attempts to parse it as a http header and then call a reply function according to the header type. If the request is of type get, the program checks whether the file is cached. If so, the program sends a head to the server to check the cached file is not obsolete, and can be sent. Else, the program must re-GET the file from the server and cache.

  If the request is not a GET method, the program simply relays the requests between client and server, without need for caching.

- void get_mod_time_length(char *buffer, char *time)
  A small 8 line function that finds the Last-Modified field in a received header, and returns the time as a string.

- void usage()
  Prints out a usage message

- int relay_body(int recvfd, int sendfd, char *buffer, int content_len)
  Sends the body of a response from server to client or vice versa, and uses a 4kb buffer to do so. Returns -1 on error.

- int body_to_buffer(int recvfd, char *buffer, int content_len)
  Used to write a server GET response body, to the cache. Uses a 4kb buffer to do so. Returns -1 on error.

- int recv_header(int connfd, char *buffer)
  Receives a header from either client or server, by checking for the sequence of characters '\r\n\r\n' which indicate the end of a header. If those characters don't come, the program may very well hang in this function.

- void update_lru()
  Used to increment the lru fields of all the cached items. Used for the purpose of keeping track of the least recently used cache file.

- struct cache_file {
  ```
  char *buffer;
  char name[16];
  char time[30];
  int lru;
  struct cache_file* next;
  ```
  };
  Used as a node for a singly linked list of cache items

- struct cache_file* get_cache_file(char *filename)
  Returns a pointer to the cache file in the singly linked list with the given filename

- void add_cache_file(struct cache_file *cf)
  Adds a cache file to the head of the singly linked list of cache files.

- void remove_cache_file(char *filename)
  Removes the cache file with the given name. Has to take special care of not disturbing the list, when the given file is the tail, head, or both of the list.

- void replace_lru(struct cache_file *cf)
  LRU replacement of the cache. Replaces the cache file with the greatest lru field, with the given cache file.

- void replace_fifo(struct cache_file *cf)
  FIFO replacement of the cache. Free the tail, and add to the head.

- Int get_content_length(char *buffer)
  A small 8 line function that finds the content-length field in a received header, and returns the content length. If no content length, it returns -1.

QUESTIONS

For Assignment 3, please answer the following questions:

●Using a large file (e.g. 100 MiB — adjust according to your computer's capacity) and the provided HTTP server:

○Start the server with only one thread in the same directory as the large file (so that it can provide it to requests);

○Start your proxy with no cache and request the file ten times. How long does it take?

Requesting the file ten times took about 44 seconds.

○Now stop your proxy and start again, this time with cache enabled for that file.Request the same file ten times. How long does it take?

Requesting the file ten times still took about 44 seconds!? Maybe I should use an even larger file?

●Aside from caching, what other uses can you consider for a reverse proxy?

A reverse proxy could be used for client dispatching. As in, a client first connects to a reverse proxy, which then routes the client to the server that the proxy thinks the client should be interacting with. Maybe it decides this based on client location, or by the request type.

TESTING

I did all my testing using curl. Much of my testing involved around 3 different phases. First I tested the command on its ability to correctly sending/receiving put and

head requests. Then I tested Get requests using the cache, then I tested Get requests using the cache with the flags enabled. I would send mostly small files, as this was better for quicker testing, and I had already seen that the server was successful at sending very large files. What took the most time was dealing with successful use of the cache, as the singly linked list was giving me some trouble.